13 | 基本数据类型：为什么Go要原生支持字符串类型？

2021-11-10 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 24:17 大小 22.24M

你好，我是 Tony Bai。

在上节课中，我们讲解了在 Go 编程中最广泛使用的一类基本数据类型：**数值类型**，包括整型、浮点类型和复数类型。这一节课，我们继续来学习 Go 语言中另一类基本数据类型：**字符串类型**。

领资料

字符串类型，是现代编程语言中最常用的数据类型之一，多数主流编程语言都提供了对这个类型的原生支持，少数没有提供原生字符串的类型的主流语言（比如 C 语言）也通过其他形式提供了对字符串的支持。

对于这样在日常开发中高频使用的基本数据类型，我们要给予更多的关注。所以，我们这一节课，将会按照 Why-What-How 的逻辑，讲清楚 Go 对字符串类型的支持，让你对

Go 语言中的字符串有个完整而清晰的认识。

首先，让我们来看看为什么 Go 要原生支持字符串类型。

原生支持字符串有什么好处？

我们前面提过，Go 是站在巨人的肩膀上成长起来的现代编程语言。它继承了前辈语言的优点，又改进了前辈语言中的不足。这其中一处就体现在 Go 对字符串类型的原生支持上。

这样的设计会有什么好处呢？作为对比，我们先来看看前辈语言之一的 C 语言对字符串的支持情况。

C 语言没有提供对字符串类型的原生支持，也就是说，C 语言中并没有“字符串”这个数据类型。在 C 语言中，字符串是以字符串字面值或以’\0’结尾的字符类型数组来呈现的，比如下面代码：

复制代码

```
#define GO_SLOGAN "less is more"
const char * s1 = "hello, gopher"
char s2[] = "I love go"
```

这样定义的非原生字符串在使用过程中会有很多问题，比如：

不是原生类型，编译器不会对它进行类型校验，导致类型安全性差；

字符串操作时要时刻考虑结尾的’\0’，防止缓冲区溢出；

以字符数组形式定义的“字符串”，它的值是可变的，在并发场景中需要考虑同步问题；

获取一个字符串的长度代价较大，通常是 O(n) 时间复杂度；

C 语言没有内置对非 ASCII 字符（如中文字符）的支持。

这些问题都大大加重了开发人员在使用字符串时的心智负担。于是，Go 设计者们选择了原生支持字符串类型。

在 Go 中，字符串类型为 **string**。Go 语言通过 string 类型统一了对“字符串”的抽象。这样无论是字符串常量、字符串变量或是代码中出现的字符串字面值，它们的类型都被统一设置为 string，比如上面 C 代码换成等价的 Go 代码是这样的：

```
复制代码
1 const (
2   GO_SLOGAN = "less is more" // GO_SLOGAN是string类型常量
3   s1 = "hello, gopher"       // s1是string类型常量
4 )
5
6 var s2 = "I love go" // s2是string类型变量
```

那既然我们都说了，Go 原生支持 string 的做法是对前辈语言的改进，这样的设计到底有哪些优秀的性质，会带来什么好处呢？

第一点：string 类型的数据是不可变的，提高了字符串的并发安全性和存储利用率。

Go 语言规定，字符串类型的值在它的生命周期内是不可改变的。这就是说，如果我们声明了一个字符串类型的变量，那我们是无法通过这个变量改变它对应的字符串值的，但这并不是说我们不能为一个字符串类型变量进行二次赋值。

什么意思呢？我们看看下面的代码就好理解了：

```
复制代码
1 var s string = "hello"
2 s[0] = 'k'   // 错误：字符串的内容是不可改变的
3 s = "gopher" // ok
```

在这段代码中，我们声明了一个字符串类型变量 s。当我们试图通过下标方式把这个字符串的第一个字符由 h 改为 k 的时候，我们会收到编译器错误的提示：**字符串是不可变的**。但我们仍可以像最后一行代码那样，为变量 s 重新赋值为另外一个字符串。

Go 这样的“字符串类型数据不可变”的性质给开发人员带来的最大好处，就是我们不用再担心字符串的并发安全问题。这样，Go 字符串可以被多个 Goroutine（Go 语言的轻量级用户线程，后面我们会详细讲解）共享，开发者不用因为担心并发安全问题，使用会带来一定开销的同步机制。

另外，也由于字符串的不可变性，针对同一个字符串值，无论它在程序的几个位置被使用，Go 编译器只需要为它分配一块存储就好了，大大提高了存储利用率。

第二点：没有结尾’\0’ ，而且获取长度的时间复杂度是常数时间，消除了获取字符串长度的开销。

在 C 语言中，获取一个字符串的长度可以调用标准库的 strlen 函数，这个函数的实现原理是遍历字符串中的每个字符并做计数，直到遇到字符串的结尾’\0’停止。显然这是一个线性时间复杂度的算法，执行时间与字符串中字符个数成正比。并且，它存在一个约束，那就是传入的字符串必须有结尾’\0’ ，结尾’\0’是字符串的结束标志。如果你使用过 C 语言，想必你也吃过字符串结尾’\0’的亏。

Go 语言修正了这个缺陷，Go 字符串中没有结尾’\0’ ，获取字符串长度更不需要结尾’\0’作为结束标志。并且，Go 获取字符串长度是一个常数级时间复杂度，无论字符串中字符个数有多少，我们都可以快速得到字符串的长度值。至于这方面的原理，我们等会再详细说明。

第三点：原生支持“所见即所得”的原始字符串，大大降低构造多行字符串时的心智负担。

如果我们要在 C 语言中构造多行字符串，一般就是两个方法：要么使用多个字符串的自然拼接，要么需要结合续行符""。但因为有转义字符的存在，我们很难控制好格式。**Go 语言就简单多了，**通过一对反引号原生支持构造“所见即所得”的原始字符串（Raw String）。而且，Go 语言原始字符串中的任意转义字符都不会起到转义的作用。比如下面这段代码：

复制代码

```
var s string = `          ,_---~~~~~~~~~~._
    _,,_,*^____      _____*g*\"*,--,
   / __/ /'     ^.  /      \ ^@q   f
  [  @f | @))    |  | @))   l  0 _/
   \/   \~____ / __ \_____/     \
    |           _l__l_           I
    }          [______]           I
    ]            | | |            |
    ]             ~ ~             |
    |                             |
     |                            |`
```

```
12 fmt.Println(s)
```

你可以看到，字符串变量 s 被赋值了一个由一对反引号包裹的 Gopher 图案。这个 Gopher 图案由诸多 ASCII 字符组成，其中就包括了转义字符。这个时候，如果我们通过 Println 函数输出这个字符串，得到的图案和上面的图案并无二致。

第三点：对非 ASCII 字符提供原生支持，消除了源码在不同环境下显示乱码的可能。

Go 语言源文件默认采用的是 Unicode 字符集，Unicode 字符集是目前市面上最流行的字符集，它囊括了几乎所有主流非 ASCII 字符（包括中文字符）。Go 字符串中的每个字符都是一个 Unicode 字符，并且这些 Unicode 字符是以 UTF-8 编码格式存储在内存当中的。在接下来讲解 Go 字符串的组成时，我们还会对这部分内容做进一步讲解。

知道了 Go 原生支持字符串类型带来的诸多变化和好处后，我们接下来就要深入到 Go 字符串的机制里看看，看看 Go 字符串是由什么组成的。

Go 字符串的组成

Go 语言在看待 Go 字符串组成这个问题上，有两种视角。一种是**字节视角**，也就是和所有其它支持字符串的主流语言一样，**Go 语言中的字符串值也是一个可空的字节序列，字节序列中的字节个数称为该字符串的长度。一个个的字节只是孤立数据，不表意。**

比如在下面代码中，我们输出了字符串中的每个字节，以及整个字符串的长度：

复制代码

```
1 var s = "中国人"
2 fmt.Printf("the length of s = %d\n", len(s)) // 9
3
4 for i := 0; i < len(s); i++ {
5   fmt.Printf("0x%x ", s[i]) // 0xe4 0xb8 0xad 0xe5 0x9b 0xbd 0xe4 0xba 0xba
6 }
7 fmt.Printf("\n")
```

我们看到，由“中国人”构成的字符串的字节序列长度为 9。并且，仅从某一个输出的字节来看，它是不能与字符串中的任一个字符对应起来的。

如果要表意，我们就需要从字符串的另外一个视角来看，也就是**字符串是由一个可空的字符序列构成**。这个时候我们再看下面代码：

复制代码

```
var s = "中国人"
fmt.Println("the character count in s is", utf8.RuneCountInString(s)) // 3

for _, c := range s {
  fmt.Printf("0x%x ", c) // 0x4e2d 0x56fd 0x4eba
}
fmt.Printf("\n")
```

在这段代码中，我们输出了字符串中的字符数量，也输出了这个字符串中的每个字符。前面说过，Go 采用的是 Unicode 字符集，每个字符都是一个 Unicode 字符，那么这里输出的 0x4e2d、0x56fd 和 0x4eba 就应该是某种 Unicode 字符的表示了。没错，以 0x4e2d 为例，它是汉字“中”在 Unicode 字符集表中的码点（Code Point）。

那么，什么是 Unicode 码点呢？

Unicode 字符集中的每个字符，都被分配了统一且唯一的字符编号。所谓 **Unicode 码点**，就是指将 Unicode 字符集中的所有字符“排成一队”，字符在这个“队伍”中的**位次**，就是它在 Unicode 字符集中的码点。也就说，一个码点唯一对应一个字符。“码点”的概念和我们马上要讲的 rune 类型有很大关系。

rune 类型与字符字面值

Go 使用 rune 这个类型来表示一个 Unicode 码点。rune 本质上是 int32 类型的别名类型，它与 int32 类型是完全等价的，在 Go 源码中我们可以看到它的定义是这样的：

复制代码

```
// $GOROOT/src/builtin.go
type rune = int32
```

由于一个 Unicode 码点唯一对应一个 Unicode 字符。所以我们可以说，**一个 rune 实例就是一个 Unicode 字符，一个 Go 字符串也可以被视为 rune 实例的集合。我们可以通过字符字面值来初始化一个 rune 变量。**

在 Go 中，字符字面值有多种表示法，最常见的是**通过单引号括起的字符字面值**，比如：

复制代码

```
'a'  // ASCII字符
'中' // Unicode字符集中的中文字符
'\n' // 换行字符
'\'' // 单引号字符
```

我们还可以使用 **Unicode 专用的转义字符\u 或\U 作为前缀，**来表示一个 Unicode 字符，比如：

复制代码

```
'\u4e2d'      // 字符：中
'\U00004e2d' // 字符：中
'\u0027'      // 单引号字符
```

这里，我们要注意，\u 后面接两个十六进制数。如果是用两个十六进制数无法表示的 Unicode 字符，我们可以使用\U，\U 后面可以接四个十六进制数来表示一个 Unicode 字符。

而且，由于表示码点的 rune 本质上就是一个整型数，所以我们还可**用整型值来直接作为字符字面值给 rune 变量赋值**，比如下面代码：

复制代码

```
'\x27'  // 使用十六进制表示的单引号字符
'\047'  // 使用八进制表示的单引号字符
```

字符串字面值

字符串是字符的集合，了解了字符字面值后，字符串的字面值也就很简单了。只不过字符串是多个字符，所以我们需要**把表示单个字符的单引号，换为表示多个字符组成的字符串的双引号**就可以了。我们可以看下面这些例子：

复制代码

```
"abc\n"
"中国人"
```

```
"\u4e2d\u56fd\u4eba" // 中国人
"\U00004e2d\U000056fd\U00004eba" // 中国人
"中\u56fd\u4eba" // 中国人，不同字符字面值形式混合在一起
"\xe4\xb8\xad\xe5\x9b\xbd\xe4\xba\xba" // 十六进制表示的字符串字面值：中国人
```

我们看到，将单个 Unicode 字符字面值一个接一个地连在一起，并用双引号包裹起来就构成了字符串字面值。甚至，我们也可以像倒数第二行那样，将不同字符字面值形式混合在一起，构成一个字符串字面值。

不过，这里你可能发现了一个问题，上面示例代码的最后一行使用的是十六进制形式的字符串字面值，但每个字节的值与前面几行的码点值完全对应不上啊，这是为什么呢？

这个字节序列实际上是“中国人”这个 Unicode 字符串的 UTF-8 编码值。什么是 UTF-8 编码？它又与 Unicode 字符集有什么关系呢？

UTF-8 编码方案

UTF-8 编码解决的是 Unicode 码点值在计算机中如何存储和表示（位模式）的问题。那你可能会说，码点唯一确定一个 Unicode 字符，直接用码点值不行么？

这的确是可以的，并且 UTF-32 编码标准就是采用的这个方案。UTF-32 编码方案固定使用 4 个字节表示每个 Unicode 字符码点，这带来的好处就是编解码简单，但缺点也很明显，主要有下面几点：

- 这种编码方案使用 4 个字节存储和传输一个整型数的时候，需要考虑不同平台的字节序问题；
- 由于采用 4 字节的固定长度编码，与采用 1 字节编码的 ASCII 字符集无法兼容；
- 所有 Unicode 字符码点都用 4 字节编码，显然空间利用率很差。

针对这些问题，Go 语言之父 Rob Pike 发明了 UTF-8 编码方案。和 UTF-32 方案不同，UTF-8 方案使用变长度字节，对 Unicode 字符的码点进行编码。编码采用的字节数量与 Unicode 字符在码点表中的序号有关：表示序号（码点）小的字符使用的字节数量少，表示序号（码点）大的字符使用的字节数多。

UTF-8 编码使用的字节数量从 1 个到 4 个不等。前 128 个与 ASCII 字符重合的码点（U+0000~U+007F）使用 1 个字节表示；带变音符号的拉丁文、希腊文、西里尔字母、阿拉伯文等使用 2 个字节来表示；而东亚文字（包括汉字）使用 3 个字节表示；其他极少使用的语言的字符则使用 4 个字节表示。

这样的编码方案是兼容 ASCII 字符内存表示的，这意味着采用 UTF-8 方案在内存中表示 Unicode 字符时，已有的 ASCII 字符可以被直接当成 Unicode 字符进行存储和传输，不用再做任何改变。

此外，UTF-8 的编码单元为一个字节（也就是一次编解码一个字节），所以我们在处理 UTF-8 方案表示的 Unicode 字符的时候，就不需要像 UTF-32 方案那样考虑字节序问题了。相对于 UTF-32 方案，UTF-8 方案的空间利用率也是最高的。

现在，UTF-8 编码方案已经成为 Unicode 字符编码方案的事实标准，各个平台、浏览器等默认均使用 UTF-8 编码方案对 Unicode 字符进行编、解码。Go 语言也不例外，采用了 UTF-8 编码方案存储 Unicode 字符，我们在前面按字节输出一个字符串值时看到的字节序列，就是对字符进行 UTF-8 编码后的值。

那么现在我们就使用 Go 在标准库中提供的 UTF-8 包，对 Unicode 字符（rune）进行编解码试试看：

复制代码

```
// rune -> []byte
func encodeRune() {
    var r rune = 0x4E2D
    fmt.Printf("the unicode charactor is %c\n", r) // 中
    buf := make([]byte, 3)
    _ = utf8.EncodeRune(buf, r) // 对rune进行utf-8编码
    fmt.Printf("utf-8 representation is 0x%X\n", buf) // 0xE4B8AD
}

// []byte -> rune
func decodeRune() {
    var buf = []byte{0xE4, 0xB8, 0xAD}
    r, _ := utf8.DecodeRune(buf) // 对buf进行utf-8解码
    fmt.Printf("the unicode charactor after decoding [0xE4, 0xB8, 0xAD] is %s\
}
```

这段代码中，encodeRune 通过调用 UTF-8 的 EncodeRune 函数实现了对一个 rune，也就是一个 Unicode 字符的编码，decodeRune 则调用 UTF-8 包的 decodeRune，将一段内存字节转换回一个 Unicode 字符。

好了，现在我们已经搞清楚 Go 语言中字符串类型的性质和组成了。有了这些基础之后，我们就可以看看 Go 是如何实现字符串类型的。也就是说，在 Go 的编译器和运行时中，一个字符串变量究竟是如何表示的？

Go 字符串类型的内部表示

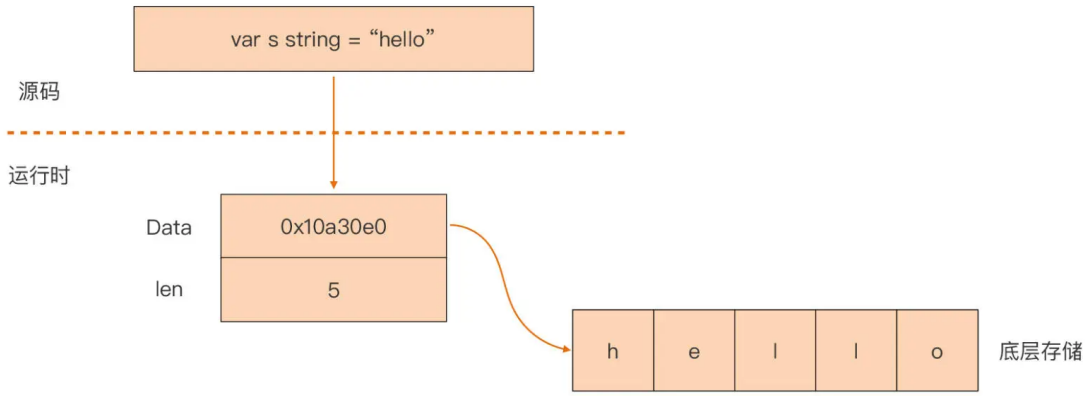
其实，我们前面提到的 Go 字符串类型的这些优秀的性质，Go 字符串在编译器和运行时中的内部表示是分不开的。Go 字符串类型的内部表示究竟是什么样的呢？在标准库的 reflect 包中，我们找到了答案，你可以看看下面代码：

复制代码

```
// $GOROOT/src/reflect/value.go

// StringHeader是一个string的运行时表示
type StringHeader struct {
    Data uintptr
    Len  int
}
```

我们可以看到，**string 类型其实是一个“描述符”，它本身并不真正存储字符串数据，而仅是由一个指向底层存储的指针和字符串的长度字段组成的。**我也画了一张图，直观地展示了一个 string 类型变量在 Go 内存中的存储：



极客时间

你看，Go 编译器把源码中的 string 类型映射为运行时的一个二元组（Data, Len），真实的字符串值数据就存储在一个被 Data 指向的底层数组中。通过 Data 字段，我们可以得到这个数组的内容，你可以看看下面这段代码：

复制代码

```
func dumpBytesArray(arr []byte) {
    fmt.Printf("[")
    for _, b := range arr {
        fmt.Printf("%c ", b)
    }
    fmt.Printf("]\n")
}

func main() {
    var s = "hello"
    hdr := (*reflect.StringHeader)(unsafe.Pointer(&s)) // 将string类型变量地址显
    fmt.Printf("0x%x\n", hdr.Data) // 0x10a30e0
    p := (*[5]byte)(unsafe.Pointer(hdr.Data)) // 获取Data字段所指向的数组的指针
    dumpBytesArray((*p)[:]) // [h e l l o ]   // 输出底层数组的内容
}
```

这段代码利用了 unsafe.Pointer 的通用指针转型能力，按照 StringHeader 给出的结构内存布局，“顺藤摸瓜”，一步步找到了底层数组的地址，并输出了底层数组内容。

知道了 string 类型的实现原理后，我们再回头看看 Go 字符串类型性质中“获取长度的时间复杂度是常数时间”那句，是不是就很好理解了？之所以是常数时间，那是因为字符串类型中包含了字符串长度信息，当我们用 len 函数获取字符串长度时，len 函数只要简单地将这个信息提取出来就可以了。

了解了 string 类型的实现原理后，我们还可以得到这样一个结论，那就是**我们直接将 string 类型通过函数 / 方法参数传入也不会带来太多的开销。**因为传入的仅仅是一个“描述符”，而不是真正的字符串数据。

那么，了解了 Go 字符串的一些基本信息和原理后，我们从理论转向实际，看看日常开发中围绕字符串类型都有哪些常见操作。

Go 字符串类型的常见操作

由于字符串的不可变性，针对字符串，我们更多是尝试对其进行读取，或者将它作为一个组成单元去构建其他字符串，又或是转换为其他类型。下面我们逐一来看一下这些字符串操作：

第一个操作：下标操作。

在字符串的实现中，真正存储数据的是底层的数组。字符串的下标操作本质上等价于底层数组的下标操作。我们在前面的代码中实际碰到过针对字符串的下标操作，形式是这样的：

复制代码

```
1 var s = "中国人"
2 fmt.Printf("0x%x\n", s[0]) // 0xe4：字符“中” utf-8编码的第一个字节
```

我们可以看到，通过下标操作，我们获取的是字符串中特定下标上的字节，而不是字符。

第二个操作：字符迭代。

Go 有两种迭代形式：常规 for 迭代与 for range 迭代。**你要注意，通过这两种形式的迭代对字符串进行操作得到的结果是不同的。**

通过常规 for 迭代对字符串进行的操作是一种字节视角的迭代，每轮迭代得到的的结果都是组成字符串内容的一个字节，以及该字节所在的下标值，这也等价于对字符串底层数组的迭代，比如下面代码：

复制代码

```
var s = "中国人"

for i := 0; i < len(s); i++ {
  fmt.Printf("index: %d, value: 0x%x\n", i, s[i])
}
```

运行这段代码，我们会看到，经过常规 for 迭代后，我们获取到的是字符串里字符的 UTF-8 编码中的一个字节：

复制代码

```
index: 0, value: 0xe4
index: 1, value: 0xb8
index: 2, value: 0xad
index: 3, value: 0xe5
index: 4, value: 0x9b
index: 5, value: 0xbd
index: 6, value: 0xe4
index: 7, value: 0xba
index: 8, value: 0xba
```

而像下面这样使用 for range 迭代，我们得到的又是什么呢？我们继续看代码：

复制代码

```
var s = "中国人"

for i, v := range s {
    fmt.Printf("index: %d, value: 0x%x\n", i, v)
}
```

同样运行一下这段代码，我们得到：

复制代码

```
index: 0, value: 0x4e2d
index: 3, value: 0x56fd
```

```
3 index: 6, value: 0x4eba
```

我们看到，通过 for range 迭代，我们每轮迭代得到的是字符串中 Unicode 字符的码点值，以及该字符在字符串中的偏移值。我们可以通过这样的迭代，获取字符串中的字符个数，而通过 Go 提供的内置函数 len，我们只能获取字符串内容的长度（字节个数）。当然了，获取字符串中字符个数更专业的方法，是调用标准库 UTF-8 包中的 RuneCountInString 函数，这点你可以自己试一下。

第三个操作：字符串连接。

我们前面已经知道，字符串内容是不可变的，但这并不妨碍我们基于已有字符串创建新字符串。Go 原生支持通过 +/+= 操作符进行字符串连接，这也是对开发者体验最好的字符串连接操作，你可以看看下面这段代码：

复制代码

```
1 s := "Rob Pike, "
2 s = s + "Robert Griesemer, "
3 s += " Ken Thompson"
4
5 fmt.Println(s) // Rob Pike, Robert Griesemer, Ken Thompson
```

不过，虽然通过 +/+= 进行字符串连接的开发体验是最好的，但连接性能就未必是最快的了。除了这个方法外，Go 还提供了 strings.Builder、strings.Join、fmt.Sprintf 等函数来进行字符串连接操作。关于这些方法的性能讨论，我放到了后面的思考题里，我想让你先去找一下答案。

第四个操作：字符串比较。

Go 字符串类型支持各种比较关系操作符，包括 = =、!= 、>=、<=、> 和 <。在字符串的比较上，Go 采用字典序的比较策略，分别从每个字符串的起始处，开始逐个字节地对两个字符串类型变量进行比较。

当两个字符串之间出现了第一个不相同的元素，比较就结束了，这两个元素的比较结果就会做为串最终的比较结果。如果出现两个字符串长度不同的情况，长度比较小的字符串会用空元素补齐，空元素比其他非空元素都小。

这里我给了一个 Go 字符串比较的示例：

复制代码

```
func main() {
        // ==
        s1 := "世界和平"
        s2 := "世界" + "和平"
        fmt.Println(s1 == s2) // true

        // !=
        s1 = "Go"
        s2 = "C"
        fmt.Println(s1 != s2) // true

        // < and <=
        s1 = "12345"
        s2 = "23456"
        fmt.Println(s1 < s2)  // true
        fmt.Println(s1 <= s2) // true

        // > and >=
        s1 = "12345"
        s2 = "123"
        fmt.Println(s1 > s2)  // true
        fmt.Println(s1 >= s2) // true
}
```

你可以看到，鉴于 Go string 类型是不可变的，所以说如果两个字符串的长度不相同，那么我们不需要比较具体字符串数据，也可以断定两个字符串是不同的。但是如果两个字符串长度相同，就要进一步判断，数据指针是否指向同一块底层存储数据。如果还相同，那么我们可以说两个字符串是等价的，如果不同，那就还需要进一步去比对实际的数据内容。

第五个操作：字符串转换。

在这方面，Go 支持字符串与字节切片、字符串与 rune 切片的双向转换，并且这种转换无需调用任何函数，只需使用显式类型转换就可以了。我们看看下面代码：

复制代码

```
var s string = "中国人"

// string -> []rune
```

```
rs := []rune(s)
fmt.Printf("%x\n", rs) // [4e2d 56fd 4eba]

// string -> []byte
bs := []byte(s)
fmt.Printf("%x\n", bs) // e4b8ade59bbde4baba

// []rune -> string
s1 := string(rs)
fmt.Println(s1) // 中国人

// []byte -> string
s2 := string(bs)
fmt.Println(s2) // 中国人
```

这样的转型看似简单，但无论是 string 转切片，还是切片转 string，这类转型背后也是有着一定开销的。这些开销的根源就在于 string 是不可变的，运行时要为转换后的类型分配新内存。

小结

好了，今天的课讲到这里就结束了。这一节课，我们学习了 Go 中另外一类最常用的基本数据类型：字符串类型。Go 原生支持字符串类型，所有字符串变量、常量、字面值都统一设置为 string 类型，对 string 的原生支持使得 Go 字符串有了很多优秀性质。

我们可以使用两个视角来看待 Go 字符串的组成，一种是字节视角。Go 字符串是由一个可空的字节序列组成，字节的个数称为字符串的长度；另外一种是字符视角。Go 字符串是由一个可空的字符序列构成。Go 字符串中的每个字符都是一个 Unicode 字符。

Go 使用 rune 类型来表示一个 Unicode 字符的码点。为了传输和存储 Unicode 字符，Go 还使用了 UTF-8 编码方案，UTF-8 编码方案使用变长字节的编码方式，码点小的字符用较少的字节编码，码点大的字符用较多字节编码，这种编码方式兼容 ASCII 字符集，并且拥有很高的空间利用率。

Go 语言在运行时层面通过一个二元组结构（Data, Len）来表示一个 string 类型变量，其中 Data 是一个指向存储字符串数据内容区域的指针值，Len 是字符串的长度。因此，本质上，一个 string 变量仅仅是一个“描述符”，并不真正包含字符串数据。因此，我们即便直接将 string 类型变量作为函数参数，其传递的开销也是恒定的，不会随着字符串大小的变化而变化。

Go 为其原生支持的 string 类型提供了许多原生操作类型，在进行字符串操作时你要注意以下几点：

通过常规 for 迭代与 for range 迭代所得到的结果不同，常规 for 迭代采用的是字节视角；而 for range 迭代采用的是字符视角；

基于 +/+= 操作符的字符串连接是对开发者体验最好的字符串连接方式，但却不是性能最好的方式；

无论是字符串转切片，还是切片转字符串，都会有内存分配的开销，这缘于 Go 字符串数据内容不可变的性质。

思考题

我们前面讲到，Go 提供多种字符串连接服务，包括基于 +/+= 的字符连接、基于 strings.Builder、strings.Join、fmt.Sprintf 等函数来进行字符串连接操作。那么，哪种连接方式是性能最高的呢？期待在留言区看到你的想法。

欢迎把这节课分享给更多对 Go 语言字符串类型感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 6 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 基本数据类型：Go原生支持的数值类型有哪些？

下一篇 14 | 常量：Go在“常量”设计上的创新有哪些？

更多学习推荐

2021 最新大厂
Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取

精选留言 (19)

写留言

2021-11-11

```
func plusConcat(n int, str string) string {
    // +号拼接
}

func sprintfConcat(n int, str string) string {...
```

展开

作者回复: 简直就是标准答案👍

9

你好呀, 朋友.

2021-11-12

是不是可以理解成[]rune里存的是Unicode码点或者说UTF-32编码，而[]byte和string存的是UTF-8编码

作者回复: 一个rune存储一个unicode码点或utf-32的四字节编码；从字节视角，string对应的底层存储存放的是utf8编码。

3

布凡

2021-11-10

strings.Builder的效率要比+/+=的效率高
因为string.Builder 是先将第一个字符串的地址取出来，然后将builder的字符串拼接到后面，
func (b *Builder) copyCheck() {
if b.addr == nil {...

展开

作者回复: 点个赞。

2

Aaron Liu

2021-11-10

这是见过讲string最详细的，原理，实操，对比，背后的设计以及常用方法，很透彻，继续学习

2

lesserror

2021-11-11

Tony Bai 老师的这篇关于Go字符串类型的讲解非常细致。

但还是有以下困惑，麻烦老师看到了解答一下：

1. 怎么理解：“字面值”比较贴切？...

展开

作者回复: 1. 字面值(literal)就是源码中的一个固定的值，它直接写在源码中，不可变，且不需经过任何计算我们就能从字面上看出其“值”。在编程语>言中，通常一个字面值都可以从其字面上大致推断出其类型。另外字面值可以用于初始化变量，也可以作为常量的值。

2. unsafe包属于高级话题，本专栏作为入门专栏后续也不会太过深入。
3. 嗯，用s:="中国人"，估计编辑器就不会提示了
4. 好建议！

1

qinsi
2021-11-11

那么问题来了，raw string里要怎么使用反引号？

展开

作者回复: 反引号是唯一的“漏网之鱼”:)。

1

Geek_99b47c
2021-11-16

“了解了 string 类型的实现原理后，我们还可以得到这样一个结论，那就是我们直接将 string 类型通过函数 / 方法参数传入也不会带来太多的开销。因为传入的仅仅是一个“描述符”，而不是真正的字符串数据。”

func main() {...

展开

aoe
2021-11-13

这才是最靓的仔

```
          ,_---~~~~~----._
_,,_,*^____      _____*g*\"*,--,
/ __/ /' ^. / \ ^@q f
[ @f | @)) | | @)) l 0 _/...
```

展开

共 1 条评论 >

Calvin
2021-11-12

老师您好，想请教下 Go 中的 == 字符串比较和 Java 中的有什么区别吗？
Java 中好像是字符串内存地址的比较，因为 java.lang.String 是一个对象。

作者回复: java中==是比较是否是同一对象。若要比内容，应该用equals方法。

共 3 条评论 >

豆腐居士
2021-11-11

催更。。。

展开

Calvin

2021-11-11

老师，我复制并试了文章中的多行字符串例子，为什么报错的呢？

[setup failed]
string_test.go:105:26: expected ';', found `*g*\"*,
/ __/ /' ^. / \ ^@q f ...

展开

作者回复: 原文有问题，已更正，请用最新内容重试。

2021-11-10

字符串是很常用的东西，不由得在想，go应该也有字符串乱码的问题吧？

作者回复: 实话实说，这个多年了，我还没遇到过。除非你的环境使用的不是utf8编码方式。

高雪斌

2021-11-10

+，+= 和 fmt.Sprintf效率应该是最差的，strings.Builder和strings.Join性能应该不错，特别是strings.Join

酥宝话不多

2021-11-10

不加锁的性能强

展开

在下宝龙、

2021-11-10

老师您好，一个中文字 在utf-8编码之后是三个字节 ，那为什么会没有字节序问题，我有点弄不明白

作者回复: utf8是变长编码，其编码单元是单个字节，不存在谁在高位、谁在低位的问题。而utf-16的编码单元是双字节，utf-32编码单元为4字节，均需要考虑字节序问题。

羊羊
2021-11-10

第二点：没有结尾'\0'，而且获取长度的时间复杂度是常数时间，消除了获取字符串长度的开销。这里应该是获取字符串字节长度的时间复杂度是O(1)么？如果使用utf8的库函数utf8.RuneCountInString()获取字符串字符长度，不知道时间复杂度是多少呢

展开

作者回复: 1. 获取字符串字节长度的时间复杂度是O(1)么？ - 是的
2. 使用utf8的库函数utf8.RuneCountInString()获取字符串字符长度的时间复杂度应该是o(n)。

共 2 条评论 >

羊羊
2021-11-10

var s string = "hello"
s[0] = 'k'
s = "gopher"
根据后面的内容知道s[0]是字符串s的第一个字节，不是第一个字符，是不是不能把字符k直接赋值给s[0]呢？go的字符串是不是，不能用index来获取其中的单个字符？

展开

作者回复: s[0] = 'k' 这个肯定不可以。go string是不可更改的。
go字符串下标操作只能获得第一个字节，而不是第一个字符。

罗杰
2021-11-10

关于拼接性能，有没有权威的文章介绍，我对比测试发现的结果 fmt.Sprintf 基本都是最差的

作者回复: 这个没啥权威文章，自己写benchmark跑一下就大体能判断出那个性能更好了。

功夫熊猫

2021-11-10

我感觉是直接拿加号快，因为利用函数还需要调用外部函数，这样会浪费时间

作者回复: 建议自己写一个benchmark test，跑一下，验证一下。